Coral: Cost-Efficient Multi-LLM Serving over Heterogeneous Cloud GPUs

Yixuan Mei
yixuanm@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Zikun Li
zikunl@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Zixuan Chen
zixuanc3@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Shiqi Pan
shiqip@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Mengdi Wu
mengdiwu@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Xupeng Miao*
xupeng.miao@pku.edu.cn
Peking University
Beijing, China

Zhihao Jia
zhihao@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

K. V. Rashmi
rvinayak@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

arXiv:2605.04357v2 [cs.DC] 17 Oct 2026

Abstract

The landscape of large language models (LLMs) has become increasingly fragmented, with no single model dominating the field. Meanwhile, cloud providers offer a diverse array of mid-tier and older-generation GPUs. These heterogeneous resources provide better availability and deliver comparable performance-per-dollar to top-tier hardware. To efficiently harness these GPUs for concurrent multi-LLM serving, we introduce Coral: an adaptive, heterogeneity-aware serving system. The key idea behind Coral is the joint optimization of resource allocation and replica serving strategies across all models. To keep pace with shifting throughput demands and resource availability, Coral employs a lossless, two-stage decomposition. This approach preserves joint optimality while reducing online solve times from hours to mere tens of seconds. Evaluations across 6 models and 20 GPU configurations demonstrate that Coral reduces serving costs by up to 2.79× compared to the strongest baseline, while delivering up to 2.39× higher goodput under scarce resource constraints.

1 Introduction

Large language models (LLMs) now power an expanding array of tasks, including interactive chatbots [10, 38], automated code generation [4, 7, 42], and agentic workflows [49, 52]. Yet, no single model dominates the ecosystem. Models from different families excel at distinct tasks [5, 22, 27], and models of varying sizes within those families target different cost–quality trade-offs [18, 31, 51]. Industry data underscores this fragmentation: no single model handles more than 25% of queries [40]. As a result, providers must efficiently serve multiple models concurrently. Perplexity, for example, serves 46 models for its answer engine [40], and Microsoft Office 365 backs its AI features with seven LLMs across three families [17].

GPU	A	G	R	Rel. Cost	Mem (GB)	BW (TB/s)	TF LOPS	Perf. Per Cost		
								Mem	BW	TF
H100	✓	✓	✓	7.6	80	3.35	989	10.5	0.44	129.8
A100	✓	✓	✓	3.5	80	2.04	312	22.8	0.58	88.9
L40S	✓	✗	✓	2.2	48	0.86	362	21.5	0.39	162.3
L4	✓	✓	✓	1.0	24	0.30	121	24.0	0.30	121.0
A10G	✓	✗	✗	1.2	24	0.60	70	19.7	0.49	57.4

Table 1. GPU specs and availabilities on AWS (A), GCP (G), and RunPod (R). Relative cost is mean hourly price normalized to L4. "Perf. Per Cost" divides metrics by relative cost.

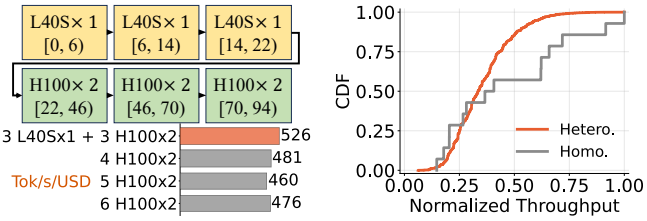
To avoid the prohibitive capital costs of dedicated AI hardware, organizations increasingly rely on public clouds to host this diverse mix of models [2, 34]. These cloud environments, however, are inherently heterogeneous, offering GPUs that span multiple generations and specifications (Table 1). While top-tier GPUs (e.g., B200, H100) deliver peak per-device capability, they are often supply-constrained. In contrast, older and mid-tier GPUs (e.g., A100, L4) are more readily available, substantially cheaper, and often yield comparable or even superior cost efficiency per unit of compute and memory.

Together, model fragmentation and cloud GPU heterogeneity motivate a new challenge: *cost-efficient, adaptive multi-LLM serving on heterogeneous cloud resources*. To address this, a serving system must take as input a set of models with latency service-level objectives (SLOs), current per-model throughput demands, and the real-time pricing and availability of GPU configurations across cloud regions. From these inputs, the system must jointly optimize two decisions: a *resource allocation* plan detailing which nodes to provision, and a *model placement* strategy dictating how to partition each model across those heterogeneous nodes. The objective

*Work done at Purdue.

Criterion	Coral	SkyS. [29]	Sage. [17]	Cauchy [55]	Helix [30]	HexG. [21]
Resource Alloc.	Joint Opt.	✓	✓	✓	✗	✗
Model Placement		✗	✗	✗	✓	✓
Latency SLO	✓	✗	✓	✓	✗	✓
Multi-LLM	✓	✗	✓	✓	✗	✗

Table 2. Comparison of heterogeneous LLM serving systems. Prior work optimizes either resource allocation [17, 29, 55] or model placement [21, 30]. Coral uniquely co-optimizes both to serve multiple LLMs under per-model latency SLOs.



(a) A mixed L40S/H100 pipeline serves Qwen-3 235B prefill more cost-efficiently than any pure-H100 setup (SLO = 1800 ms).

(b) Normalized throughput CDF for GPT-OSS 120B decode plans. Heterogeneous combinations fill gaps left by homogeneous ones.

Figure 1. Opportunities brought by heterogeneity.

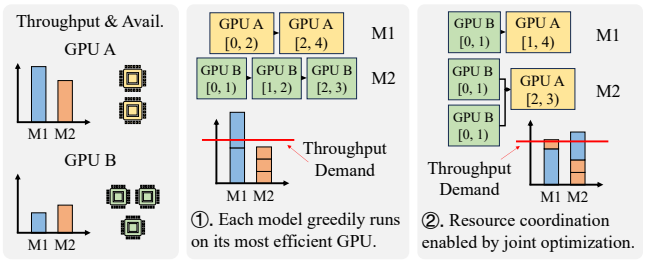


Figure 2. Joint optimization across models. Greedy per-model allocation (①) causes contention; mixed-GPU replicas produced by joint optimization (②) satisfy both demands.

is to satisfy all throughput demands and latency SLOs at the minimum total cost. Furthermore, because demand, resource availability, and pricing continuously fluctuate, the system must periodically re-solve this optimization to dynamically *adapt* the cluster to shifting conditions.

Prior work tackles this problem only in pieces. Existing heterogeneity-aware allocation systems target orthogonal goals, such as spot preemption resilience via cross-region replica spreading [29] or managing mixed interactive/batch workloads via forecast-driven scaling [17]. They share a common limitation: each model replica is treated as a black box on homogeneous hardware, abstracting model placement out of the optimization space entirely. Cauchy [55] relaxes this for prefill-decode disaggregated serving by assigning different GPU configurations to prefill and decode, but each phase remains internally homogeneous. As shown in Sec. 2.2 and Sec. 6, this results not only in significantly higher costs but also in deployment infeasibility when resources are scarce.

Helix [30] and HexGen [21] take the opposite stance, optimizing placement strictly for a *single* model on a *given* heterogeneous node set. Wrapping these single-model solvers in an outer enumeration over allocations is intractable. The allocation space is exponential, and each inner placement solve is already prohibitively expensive; Helix, for instance, takes four hours for one model on 24 nodes. As Table 2 summarizes, no existing system addresses the problem end-to-end.

Solving the heterogeneity-aware multi-LLM serving problem end-to-end poses two key challenges. *First, resource allocation and model placement are tightly coupled, and each features an exponential solution space.* The throughput and SLO attainable on a given set of nodes depend entirely on how we place the model. Conversely, the optimal placement strategy cannot be determined until the node allocation is finalized. Neither dimension can be pruned independently without prematurely constraining the other, yet a joint brute-force approach remains intractable due to the combinatorially explosive search space. *Second, optimization must complete in minutes, not hours.* Because cloud GPU availability [45, 48] and user throughput demands [44] fluctuate rapidly, a computationally expensive solution that takes hours to generate will be rendered obsolete by the time it is deployed.

To address these challenges, we introduce Coral[1]: an adaptive, heterogeneity-aware multi-LLM serving system. Coral is grounded in a key observation: for a given model and latency SLO, the throughput-optimal placement on any combination of nodes depends strictly on that combination, independent of the broader cluster allocation. Consequently, this optimal placement can be treated as a reusable artifact—which we term a *Serving Template*—that can be *precomputed offline and cached*. This insight decouples model placement from resource allocation without sacrificing joint optimality.

While the theoretical space of node combinations is unbounded, Coral enumerates a principled subset that, as demonstrated in Sec. 6.8, captures the cost-efficient regime with negligible loss. Accordingly, Coral adopts a two-stage architecture. Offline, a template generator explores node combinations for each (model, SLO) pair, leveraging an Integer Linear Program (ILP) to compute the throughput-optimal placement under SLO and produce a library of reusable Serving Templates. Online, a lightweight ILP dynamically deploys these templates across cloud regions to satisfy throughput demands at minimum cost, factoring in real-time pricing and availability. By lifting the computationally heavy placement search off the critical path, the online solver completes in

[1]Coral stands for **C**ost-efficient **O**rchestration of **R**esources for **A**daptive **L**LM-serving.

tens of seconds, ensuring the cluster remains highly adaptive to shifting demands and resource fluctuations.

We implement Coral in 53K lines of Python and C++, building a distributed runtime atop ZeroMQ [14] and NCCL [35], with vLLM [23] for per-node execution. To evaluate large-scale deployments without prohibitive hardware costs, we also develop a high-fidelity, event-driven simulator. Evaluated across 6 models, 5 GPU types, 20 GPU configurations, and varying throughput demands, Coral reduces total serving costs by up to 2.79× over the strongest baseline. Under strict resource constraints, it delivers up to 2.39× higher goodput.

In summary, our key contributions are:

- A formulation of heterogeneity-aware multi-LLM serving as a joint optimization over resource allocation and model placement under per-model latency SLOs.
- A lossless decomposition that shifts the slow model placement search offline via reusable *Serving Templates*, reducing online solve times to tens of seconds.
- ILP-based solvers for both offline Serving Template generation and online resource allocation that scale to clusters of hundreds of nodes.
- An end-to-end distributed implementation of Coral, alongside a high-fidelity simulator validated against real hardware for large-scale evaluation.
- Comprehensive evaluations across diverse models, workloads, and cluster scales, demonstrating up to 2.79× in cost savings and 2.39× higher goodput.

We will open-source the system upon paper acceptance.

2 Background and Motivation

2.1 Distributed LLM Serving

Hardware Affinity in LLM Inference. Modern LLMs use diverse Transformer architectures, including full-attention [11], hybrid-attention with sparse patterns for long contexts (e.g., sliding-window attention) [37, 46], and mixture-of-experts (MoE) [9, 18, 37, 51]. These designs fundamentally alter the balance between computation, memory capacity, and bandwidth requirements. For instance, MoE activates a sparse expert subset per token, decoupling model size from FLOPs [9, 18]. Furthermore, LLM serving operates in two distinct phases: prefill (prompt processing) and decode (auto-regressive generation) [53]. These phases exhibit contrasting hardware demands: prefill is highly parallel and compute-bound, whereas decode is sequential and heavily bottlenecked by memory bandwidth, particularly for KV-cache accesses [39, 56]. Consequently, optimal GPU selection depends jointly on model architecture and serving phase rather than a uniform hardware assignment [30, 39, 56].

Parallelism Strategies for Heterogeneous LLM Serving. Distributed LLM serving relies on four primary parallelism strategies: data (DP) [6], pipeline (PP) [16], tensor (TP) [43], and expert (EP) [24]. DP replicates models to distribute request load, while PP partitions the model into sequential layer stages [6, 16]. In contrast, TP shards individual operators across devices, and EP distributes MoE experts, routing tokens dynamically [24, 43]. DP and PP serve as effective building blocks for heterogeneous serving due to their coarse-grained partitioning: DP distributes requests proportionally, and PP adapts to varying device capabilities via uneven stage sizing [30]. However, TP and EP require fine-grained, tightly synchronized communication—such as all-reduce in TP and all-to-all exchanges in EP [24, 43]. This tight coupling makes them highly sensitive to compute and network imbalances, rendering them poorly suited for heterogeneous clusters.

2.2 Challenges and Opportunities

GPU Heterogeneity. Serving a single model replica across heterogeneous GPUs tightly couples model placement and resource allocation, creating an exponential search space. However, this complexity unlocks critical cost and performance opportunities.

Fig. 1a illustrates the most cost-efficient prefill strategy for Qwen-3 235B [51] under an 1800 ms latency SLO, using the five GPU types in Table 1. The optimal configuration is a mixed pipeline of three single-GPU L40S nodes and three dual-GPU H100 nodes with a non-uniform layer partition. Notably, this outperforms all homogeneous H100 setups (526 vs. 481, 460, and 476 Tok/s/USD). Thus, using mid-tier GPUs not only mitigates top-tier scarcity but actively *improves* cost efficiency—a pattern consistent across other models and GPU combinations. Furthermore, heterogeneous pipelines enable finer-grained capacity scaling. As Fig. 1b shows, homogeneous node sets (gray) produce discrete throughput tiers, forcing allocators to over-provision when demand falls between steps. In contrast, heterogeneous node sets (orange) generate a near-continuous throughput spectrum. This allows the system to tightly match per-model demand and eliminate wasted capacity. Together, these two effects drive the up to 2.79× cost reduction reported in Sec. 6.

Joint Optimization across LLMs. The second challenge is coordinating decisions *across* models rather than solving them in isolation. Fig. 2 demonstrates the limitations of a per-model greedy strategy. When two models (M1, M2) share a constrained pool (2 GPU-A, 3 GPU-B nodes), independent allocation (①) causes M1 to monopolize the GPU-A nodes and over-serve its demand, leaving M2 under-resourced with only GPU-B nodes. Joint optimization (②) resolves this contention: M1 accepts a mixed pipeline (one GPU-A, one GPU-B node), freeing a GPU-A node for M2 to form its own mixed replica. Consequently, both models meet their throughput demands without idle resources. Capturing these global efficiency gains requires jointly optimizing allocation and placement across *all* models—the core challenge Coral addresses.

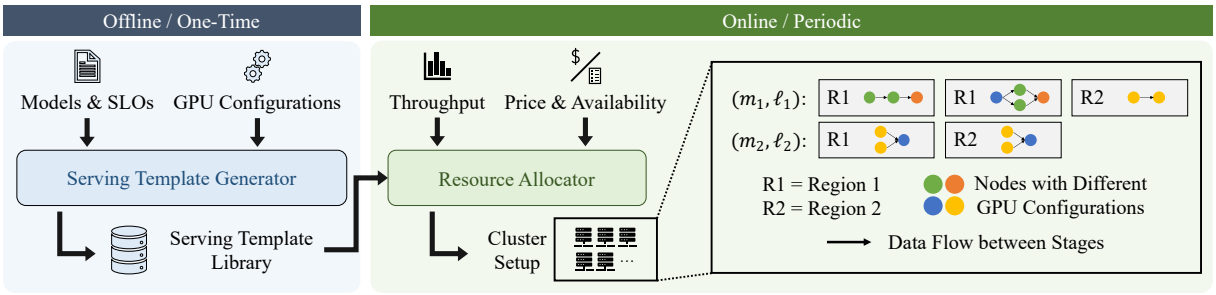


Figure 3. The two-stage workflow of Coral. *Offline*, the Serving Template generator (Sec. 4.2) pre-computes a template library from target models, SLOs, and GPU configurations. *Online*, the resource allocator (Sec. 4.3) queries this library to dynamically provision a cost-minimal, cross-region cluster that satisfies throughput demands. The offline stage runs once, enabling the online allocator to periodically adapt to workload and resource fluctuations.

3 The Multi-LLM Serving Problem

We formulate the multi-LLM serving problem on heterogeneous GPUs as a joint optimization over resource allocation and model placement. Let $\mathcal{G}$ denote the pool of available GPU nodes, where each node $g \in \mathcal{G}$ incurs a provisioning cost $p(g) \geq 0$. Let $\mathcal{M}$ be the set of models, with each model $m \in \mathcal{M}$ requiring a throughput T_m under its latency SLO.

A serving strategy is defined by two functions. First, the resource allocation Φ determines a replica count $n_m \geq 1$ for each model m and assigns each replica a disjoint subset of nodes:

$$\Phi(m) = \left(G_m^1, \ldots, G_m^{n_m}\right), \quad G_m^i \subseteq \mathcal{G},$$
$$G_m^i \cap G_{m'}^j = \emptyset \text{ for } (m,i) \neq (m',j).$$

Second, for each replica G_m^i, the model placement Ψ defines a hybrid pipeline- and data-parallel layout. It specifies the number of pipeline stages, the partition of layers across stages, and the node-to-stage mapping. Multiple nodes mapped to the same stage duplicate its layers to share the load (DP). Following prior work [21, 30], we restrict inter-node parallelism to PP and DP; tensor and expert parallelism shard work symmetrically and bottleneck at the slowest device, rendering them inefficient across heterogeneous nodes. However, TP and EP remain active intra-node, where GPUs are homogeneous and share high-bandwidth interconnects.

Let $T(\Psi(G_m^i))$ denote the throughput of model m's i-th replica under Ψ, subject to its latency SLO. The system jointly optimizes Φ and Ψ to minimize total provisioning cost while satisfying per-model throughput and latency demands:

$$\min_{\Phi,\Psi} \quad \sum_{m\in\mathcal{M}} \sum_{i=1}^{n_m} \sum_{g\in G_m^i} p(g)$$
$$\text{s.t.} \quad \sum_{i=1}^{n_m} T\left(\Psi(G_m^i)\right) \geq T_m, \quad \forall\, m \in \mathcal{M}.$$

4 Optimization Formulation in Coral

4.1 Solution Overview

Naive Enumeration is Intractable. Optimizing resource allocation (Φ) and model placement (Ψ) jointly via naive enumeration is intractable, as both subproblems exhibit exponential search spaces. At scale, this combinatorial explosion is severe. For example, Helix [30] requires a four-hour search to optimize placement for a single model on just 24 nodes, even *without* a latency SLO. Enforcing an SLO further compounds this complexity, as latency feasibility can only be verified after a full placement is finalized. Consequently, computing even a single global solution is computationally prohibitive, precluding the system from dynamically adapting to fluctuating demands and resource availability online.

Key Insight. To enable rapid online reconfiguration, Coral exploits a fundamental substructure of the joint problem: for a fixed model and latency SLO, the throughput-optimal placement $\Psi^*(\mathcal{G}')$ on a node subset $\mathcal{G}'$ depends solely on $\mathcal{G}'$, independent of the broader cluster allocation. This decoupling allows Coral to pre-compute and cache throughput-optimal placements for distinct node combinations offline. We term these reusable artifacts *Serving Templates*. While caching all combinations yields a theoretically lossless decomposition, the actual space is infinite. Therefore, Coral limits enumeration to a principled subset of combinations (Sec. 4.2), which we demonstrate has a negligible performance impact (Sec. 6.8). Crucially, this decomposition eliminates the placement search from the online critical path.

Two-Stage Workflow. Building on this insight, Coral orchestrates multi-LLM serving in two distinct stages (Fig. 3). *Offline*, the Serving Template generator (Sec. 4.2) enumerates node combinations for each model and SLO pair (m, ℓ). It employs integer linear programming (ILP) to discover the throughput-optimal placement for each node combination, populating a *Serving Template Library*. *Online*, the resource allocator (Sec. 4.3) queries this library with a lightweight ILP

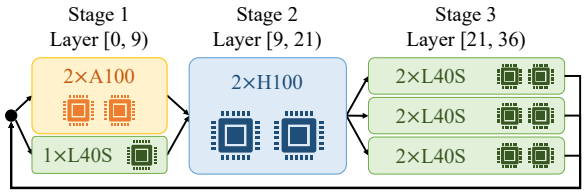


Figure 4. Serving Template. Boxes denote nodes; arrows indicate data flow. Nodes within the same pipeline stage replicate identical model layers, functioning as data-parallel replicas. During execution, a request traverses a single node per stage, looping back for auto-regressive generation.

to rapidly select and instantiate templates across regions, fulfilling global throughput demands at minimal cost while respecting current resource constraints. By moving the heavy placement search offline, online allocation resolves in tens of seconds, ensuring agile cluster adaptation.

4.2 Serving Template Generation

Serving Template. Formally, a Serving Template for model m under latency SLO ℓ is a tuple $\tau = (m, \ell, \mathcal{G}', \Psi^*(\mathcal{G}'))$. Here, $\mathcal{G}'$ is a set of nodes within a single region. Templates are strictly intra-region because inter-region network latencies (10 - 100+ milliseconds) prohibit efficient pipeline parallelism. Templates are uniquely identified by their multiset of GPU configurations. As Coral employs prefill-decode disaggregation, we generate separate templates for each phase, mapping ℓ to time-to-first-token (TTFT) or time-per-output-token (TPOT), and assuming ℓ is a fixed, known workload characteristic. $\Psi^*(\mathcal{G}')$ represents the throughput-optimal model placement for m on $\mathcal{G}'$ subject to ℓ, yielding the template's throughput $T(\tau) := T(\Psi^*(\mathcal{G}'))$. Fig. 4 illustrates a template combining 6 nodes across 4 GPU configurations into a 3-stage pipeline with intra-stage data parallelism.

Optimal Model Placement via ILP. For given node set $\mathcal{G}'$, model m with L layers, and SLO ℓ, we formulate the search for $\Psi^*(\mathcal{G}')$ as an ILP parameterized by pipeline depth S.

Decision variables. We define two sets of binary variables: $x_{sj} = 1$ iff stage $s \in [1, S]$ holds $j \in [1, L]$ consecutive layers, and $y_{sk} = 1$ iff node $g_k \in \mathcal{G}'$ is assigned to stage s.

Constraints. Each stage selects exactly one layer count, and each node maps to exactly one stage. The stage layer counts must sum to the full model size L:

$$\sum_{j=1}^{L} x_{sj} = 1 \ \forall s, \quad \sum_{s=1}^{S} y_{sk} = 1 \ \forall k, \quad \sum_{s=1}^{S} \sum_{j=1}^{L} j \cdot x_{sj} = L.$$

Throughput is bottlenecked by the slowest pipeline stage. Because nodes assigned to the same stage run data parallelism, their throughputs aggregate. For each stage s:

$$T(\tau) \ \leq \ \sum_{j=1}^{L} \sum_{k=1}^{|\mathcal{G}'|} x_{sj} \cdot y_{sk} \cdot \hat{T}_j(g_k)$$

where $\hat{T}_j(g_k)$ is the maximum throughput of node g_k hosting j layers under a per-stage latency budget of ℓ/S. This budget derives from Little's Law: for a saturated pipeline with S stages and continuous micro-batching, end-to-end latency is bounded by S times the slowest stage's latency. We determine $\hat{T}_j(g_k)$ via a one-time offline profiling step for each GPU configuration, maximizing batch size within memory limits using statistical mean input and output sequence lengths.

Linearization. To eliminate the quadratic term $x_{sj} \cdot y_{sk}$, we introduce a binary auxiliary variable z_{sjk} subject to standard McCormick envelope constraints:

$$z_{sjk} \leq x_{sj}, \qquad z_{sjk} \leq y_{sk}, \qquad z_{sjk} \geq x_{sj} + y_{sk} - 1.$$

The throughput constraint then becomes linear:

$$T(\tau) \ \leq \ \sum_{j=1}^{L} \sum_{k=1}^{|\mathcal{G}'|} z_{sjk} \cdot \hat{T}_j(g_k)$$

Objective. The ILP maximizes the end-to-end throughput $T(\tau)$. Solving this ILP for a given S yields the optimal layer partition (x_{sj}) and node-to-stage assignment (y_{sk}). The global throughput-optimal placement $\Psi^*(\mathcal{G}')$ is the configuration achieving the highest throughput across all valid pipeline depths $S \in [1, |\mathcal{G}'|]$.

This formulation contains $O(S \cdot L \cdot |\mathcal{G}'|)$ binary variables and constraints. Using modern solvers like Gurobi [13], evaluating a single S takes seconds, making a full enumeration over S highly efficient.

Generating the Serving Template Library. While preserving the original search space theoretically requires evaluating all possible node combinations, the unbounded nature of this space necessitates pruning. Coral restricts enumeration to combinations containing at most $N_{\max}$ nodes and a total GPU memory capacity below ρ times the model size. Combinations exceeding these thresholds incur high communication overheads and are typically dominated by splitting the same resources into smaller, independent replicas. Consequently, the resource allocator rarely selects them. As shown in Sec. 6.8, this pruning maintains near-optimal cost efficiency. The resulting library contains thousands to tens of thousands of templates per (m, ℓ) pair. Because each ILP solves independently, library generation is embarrassingly parallel, completing in minutes for small models and tens of minutes for larger ones (Sec. 6.8).

4.3 Online Resource Allocation

Serving Instance. To bridge offline templates and online execution, we define a *Serving Instance* as the physical realization of a template within a specific region. Each instance provisions the exact node combination $\mathcal{G}'$ dictated by its template and deploys a single model replica using the precomputed optimal layout $\Psi^*(\mathcal{G}')$. By treating Serving Instances as atomic building blocks, Coral decouples model

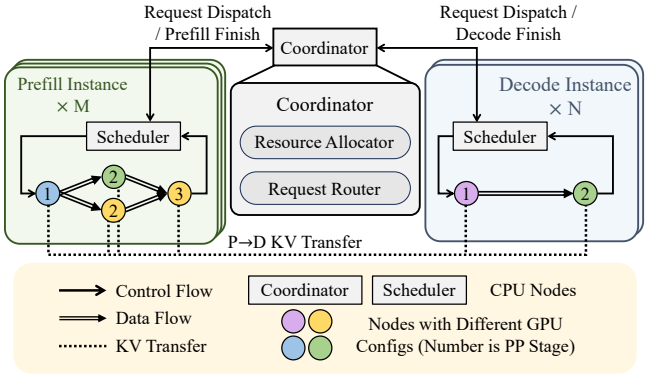


Figure 5. Coral runtime architecture. A central coordinator manages resource allocation and routes requests across prefill and decode Serving Instances. Intra-instance schedulers dispatch these requests across heterogeneous engine nodes (colored circles) per pipeline stage. KV caches transfer peer-to-peer from prefill to decode nodes.

placement from resource allocation, reducing online problem to selecting which templates to instantiate and where.

Optimal Resource Allocation via ILP. Let $\mathcal{R}$ denote the available cloud regions and C the GPU configurations. For each region $r \in \mathcal{R}$ and configuration $c \in C$, let $A_r(c) \geq 0$ be the available node count, and $p_r(c)$ the per-node provisioning cost. For a model $m \in \mathcal{M}$ requiring throughput T_m under SLO ℓ_m, the offline stage supplies N_m valid templates. Let τ_i^m denote the i-th template, $T(\tau_i^m)$ its throughput, and $U_c(\tau_i^m)$ its required number of type-c nodes. The total provisioning cost of τ_i^m in region r is $p_r(\tau_i^m) := \sum_{c \in C} p_r(c) \cdot U_c(\tau_i^m)$.

Decision variables. An integer variable $v_r(\tau_i^m) \geq 0$ dictates the number of type τ_i^m instances to deploy in region r.

Constraints. Aggregate node consumption cannot exceed regional availability for any GPU configuration c:

$$\sum_{m \in \mathcal{M}} \sum_{i=1}^{N_m} U_c(\tau_i^m) \cdot v_r(\tau_i^m) \;\leq\; A_r(c) \quad \forall r, c.$$

For each model m, total throughput must satisfy its demand:

$$\sum_{r \in \mathcal{R}} \sum_{i=1}^{N_m} T(\tau_i^m) \cdot v_r(\tau_i^m) \;\geq\; T_m \quad \forall m.$$

Objective. While Sec. 3 targets a static snapshot, the online allocator dynamically steers the cluster. Adding instances incurs non-trivial overhead (e.g., node startup, weight loading), whereas tear-down is graceful since the runtime transparently drains in-flight requests (Sec. 5.1). To penalize churn, we apply an initialization cost only to *newly added* instances. Let $v'_r(\tau_i^m)$ be the current number of running instances (a constant at solve time), and $I_r(\tau_i^m) \geq 0$ a continuous auxiliary variable bounded by:

$$I_r(\tau_i^m) \;\geq\; \big(v_r(\tau_i^m) - v'_r(\tau_i^m)\big) \cdot p_r(\tau_i^m) \cdot K,$$

where K is the ratio of node initialization time to the cluster adjustment interval. Because $I_r(\tau_i^m) \geq 0$, the bound is active only when scaling up ($v_r > v'_r$), meaning scaling down contributes nothing to the objective. The allocator minimizes the total provisioning cost plus initialization penalties:

$$\min \sum_{r \in \mathcal{R}} \sum_{m \in \mathcal{M}} \sum_{i=1}^{N_m} \Big[\, v_r(\tau_i^m) \cdot p_r(\tau_i^m) + I_r(\tau_i^m) \,\Big].$$

Tractability. This ILP introduces $2|\mathcal{R}| \cdot \sum_{m \in \mathcal{M}} N_m$ integer and continuous variables, subject to $|C| \cdot |\mathcal{R}| + |\mathcal{M}|$ physical constraints and $|\mathcal{R}| \cdot \sum_{m \in \mathcal{M}} N_m$ penalty bounds. Although variable counts can reach the millions for large template libraries, the optimal solution is exceptionally sparse: a cluster of hundreds of nodes typically activates only tens of templates. Consequently, the vast majority of $v_r(\tau_i^m)$ variables evaluate to zero, trivially satisfying their penalty bounds. As demonstrated in Sec. 6, commercial solvers like Gurobi [13] exploit this sparsity to resolve the ILP in tens of seconds, safely within our online scheduling budget.

5 Coral Runtime

5.1 System Design

Fig. 5 illustrates Coral's runtime architecture, comprising a central CPU *coordinator node* alongside pools of prefill and decode *Serving Instances*. The coordinator hosts the resource allocator (Sec. 4.3) and a request router. Each Serving Instance encapsulates a CPU *scheduler node*—managing intra-instance scheduling and coordinator communication—and a group of GPU *engine nodes* that execute model inference according to the template's model placement. While we describe Coral under prefill-decode (PD) disaggregation, this architecture natively supports PD-aggregated serving.

Request Life-Cycle. Upon request arrival, the coordinator routes it to a prefill instance via weighted round-robin, proportional to the instance's template throughput $T(\tau)$. The instance's scheduler then assigns exactly one engine node per pipeline stage, again using weighted round-robin based on each node's expected throughput under the model placement. The scheduler dispatches the prompt to the first stage, and engine nodes stream intermediate activations down the pipeline. Upon prefill completion, the scheduler notifies the coordinator, which selects a decode instance. The respective schedulers then orchestrate a direct, peer-to-peer transfer of the KV cache. Once transferred, the decode instance starts auto-regressive generation.

Instance Life-Cycle. Coral invokes the allocator periodically to reconcile the running cluster against dynamic throughput demands, resource availability, and hardware pricing.

Scaling down. When a template's target count decreases, Coral gracefully terminates the excess instances, prioritizing those with the lowest active load. A terminating instance ceases accepting new requests and shuts down only after its

in-flight requests complete (*connection draining*). Because this draining typically finishes within minutes, it is preferable to request migration; migrating in-flight decode requests requires transferring massive KV caches across the network, severely degrading system efficiency.

Scaling up. To scale up a template, Coral provisions new nodes and initializes instances according to the template's exact model placement. Dominated by node startup, weight loading, and CUDA graph compilation, this process takes several minutes—an overhead explicitly modeled by the allocator's initialization penalty (Sec. 4.3). Once fully initialized, the scheduler registers with the coordinator, and the router begins dispatching requests to the new instance.

5.2 Implementation

We implement the Coral runtime in 53K LoC of Python and C++. The control plane runs over ZeroMQ [14], on top of which we build Serving Instance life-cycle management and request dispatching. For intra-instance communication between engine nodes, we build a high-bandwidth, low-latency pipeline-parallel framework on top of NCCL [35]. This framework supports arbitrary pipeline- and data-parallel configurations and transparently enables Remote Direct Memory Access (RDMA) and GPUDirect RDMA [36] whenever the hardware permits. For KV cache transfers between prefill and decode instances, we use Gloo [32] over CPU-based RDMA. Routing the transfer through the CPU isolates KV cache movement from the GPUs, ensuring it does not contend with model inference or pipeline-parallel communication for GPU resources. We use vLLM [23] as the per-node execution engine; however, the runtime is engine-agnostic, and any compatible inference engine can be substituted without changes to the surrounding system.

To evaluate Coral at scales of hundreds of nodes—where physical deployment is cost-prohibitive—we developed an event-based simulator in 5K LoC of Python. Driven by empirical offline profiling data from our heterogeneous GPU pool, the simulator models execution at the granularity of individual pipeline stages per engine node. As Sec. 6.2 demonstrates, it reproduces real-world per-request prefill and decode latencies to within 5.6% and 7.2% respectively, confirming its high fidelity as a proxy for large-scale experiments.

6 Evaluation

We evaluate Coral to answer the following questions:

- Does Coral reduce serving costs across diverse model mixtures and heterogeneous GPU pools? (Sec. 6.3)
- Can Coral sustain high goodput and mitigate contention under tight resource availability? (Sec. 6.4)
- How resilient are Coral's cost savings to varying throughput distributions across models? (Sec. 6.5)
- Does Coral outperform SOTA heterogeneous serving (Helix) even in the single-model regime? (Sec. 6.6)

Model	MoE	Hyb. Attn.	L	Prefill (ms)	Decode (ms)	Trace
Phi4 14B	✗	✗	40	1200	60	AzureConv.
GPT-OSS 20B	✓	✓	24	900	30	AzureCode
Qwen3 32B	✗	✗	64	1600	100	BurstGPT
Llama3 70B	✗	✗	80	1500	80	BurstGPT
GPT-OSS 120B	✓	✓	36	1000	40	AzureConv.
Qwen3 235B	✓	✗	94	1800	120	AzureCode

Table 3. Model characteristics and serving metrics. First three columns: the use of mixture-of-experts (MoE) [9], hybrid sliding-window attention, and total layer count (L).

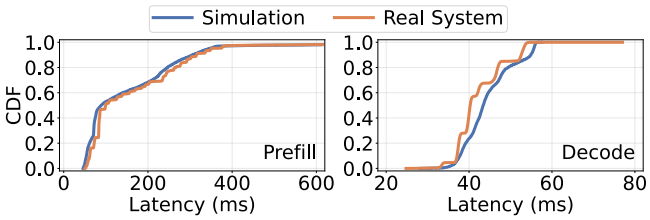


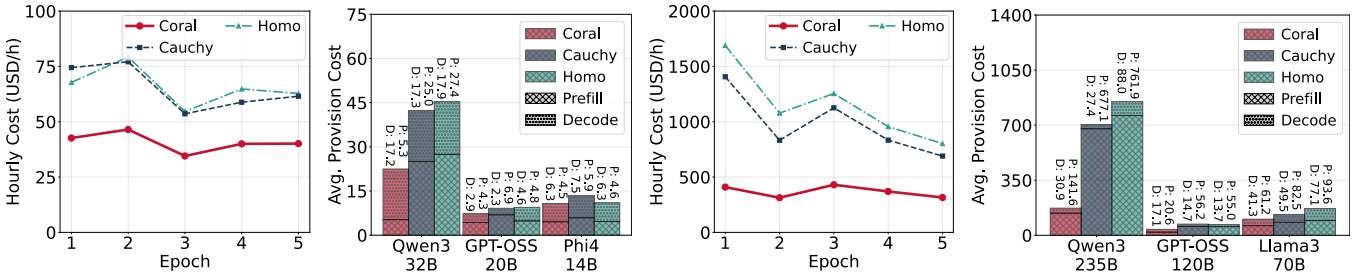
Figure 6. Simulated prefill and decode latency CDFs for Phi-4 14B tightly align with real-hardware measurements.

6.1 Experiment Setup

Model and GPU Setup. We evaluate Coral across two deployment scales. The core setup features three models (Qwen-3 32B [51], GPT-OSS 20B [37], Phi4-14B [1]) running on 12 GPU configurations (1/2/4/8× L40S, L4, A10G) across two cloud regions, requiring 20–40 GPUs. The extended setup **adds** three larger models (Qwen-3 235B [51], GPT-OSS 120B [37], Llama-3 70B [11]) and eight top-tier configurations (1/2/4/8× H100, A100), expanding to a third region and 100–300 GPUs total. Table 3 details model properties. We evaluate the core setup on real hardware (Sec. 6.3, 6.4) and in simulation (Sec. 6.5). The extended setup runs exclusively in simulation to avoid the prohibitive (thousands of dollars per hour) cost of provisioning hundreds of GPUs. Re-running the real system experiments in simulation (Sec. 6.2) yields prefill and decode latency errors of only 5.6% and 7.2% on average, confirming the simulator's high fidelity.

Workload Setup. Request lengths and arrival patterns are drawn from Azure Code [44], Azure Conversation [44], and BurstGPT [47]. We assign datasets evenly across models. Table 3 maps datasets and latency SLOs per model; SLOs are calibrated to model size and architecture (e.g., MoE [9] vs. hybrid attention [3]) following AdaServe [25]. We scale traces uniformly to achieve a default average arrival rate of 10 req/s (core setup) or 25 req/s (extended setup) per model. Sec. 6.5 evaluates skewed arrival rates, routing 80% of traffic strictly to either large or small models.

Resource Setup. GPU availability is derived from Alibaba's production cluster traces [8]. By default, we scale availability



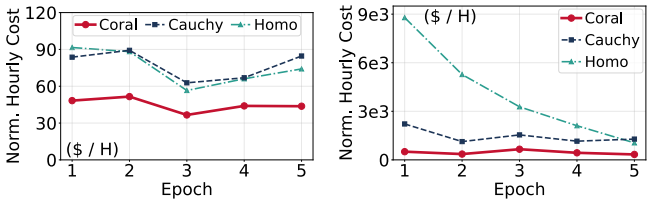
(a) Hourly cost comparison in core setup.

(b) Per-model average provision cost breakdown in core setup.

(c) Hourly cost comparison in extended setup.

(d) Per-model average provision cost breakdown in extended setup.

Figure 7. Hourly cost comparison under default settings. (a, c) Hourly cost per epoch across both deployment scales. (b, d) Per-model average provisioning cost decomposed into prefill (P) and decode (D) phases. The extended setup breakdown displays only the three largest models, which dominate total expenditure.



(a) Core setup.

(b) Extended setup.

Figure 8. Goodput-normalized hourly cost under tight resource availability.

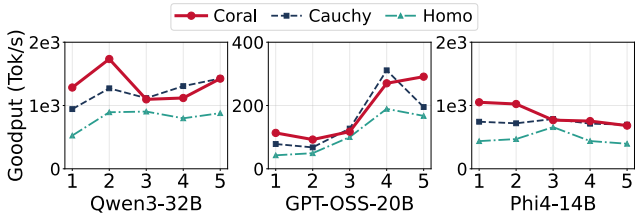


Figure 9. Decode goodput across epochs under tight resource availability (core setup). Prefill follows the same trend.

high enough to guarantee feasibility for all methods. For our resource-constrained study (Sec. 6.4), we throttle availability to a tight margin: 25% (core setup) and 75% (extended setup) above estimated minimum demand. Node pricing strictly mirrors public on-demand rates for AWS US-East-2, AWS AP-Northeast-2, and GCP US-Central-1.

Evaluation Duration. Experiments run for 30 minutes, dynamically reconfiguring the cluster every 6 minutes (one ***epoch***). We cap experiments at this duration to control physical deployment costs. Because production clusters adjust less frequently, we calculate hourly cost by amortizing initialization penalties over a standard 60-minute interval.

Coral Setup. The offline Serving Template Library is generated on a 384-core AWS c8i instance, capping templates at $N_{\max} = 6$ nodes and memory capacity at $\rho = 12\times$ model size (Sec. 6.8 analyzes sensitivity to $N_{\max}$ and ρ). This one-time process completes in a few hours. The online resource allocator executes on a 32-core c8i instance.

Baselines. We evaluate against two end-to-end baselines implemented atop the Coral runtime for fairness. **Homo** restricts each replica to homogeneous hardware, though hardware can vary across replicas (the same assumption used in SkyServe [29] and SageServe [17]). It greedily provisions the most cost-efficient (highest goodput per USD) homogeneous strategy per model. **Cauchy** is adapted from the PD-disaggregated serving system of the same name [55]. We retain its cost-efficiency model and resource-allocation algorithm, and extend its GPU-combo definition to allow a single prefill replica to feed multiple decode replicas, granting it more flexibility for multi-LLM serving. Heterogeneity-aware placement systems like **Helix** [30] do not perform resource allocation, so we compare against Helix separately in Sec. 6.6.

Evaluation Metrics. Our primary metric is *hourly cost* (USD/h), which aggregates machine provisioning and amortized initialization costs. For contention studies (Sec. 6.4), we additionally report *goodput*: the rate of generated tokens (Tok/s) that successfully meet their latency SLO.

6.2 Simulator Fidelity

To validate the simulator, we replicate all real-hardware experiments from Sec. 6.3 and Sec. 6.4. We use per-request latency as our primary fidelity metric: because aggregate metrics like goodput and SLO attainment derive directly from latency, matching the underlying latency distribution guarantees accuracy across any SLO threshold. Across all setups, simulated prefill and decode latencies deviate from real hardware by an average of only 5.6% and 7.2%, respectively. Fig. 6 illustrates a representative comparison for Phi-4 14B, demonstrating tight alignment across the full latency CDF, including the tail. We further validate the most cost-efficient Serving Templates for all three large models introduced in

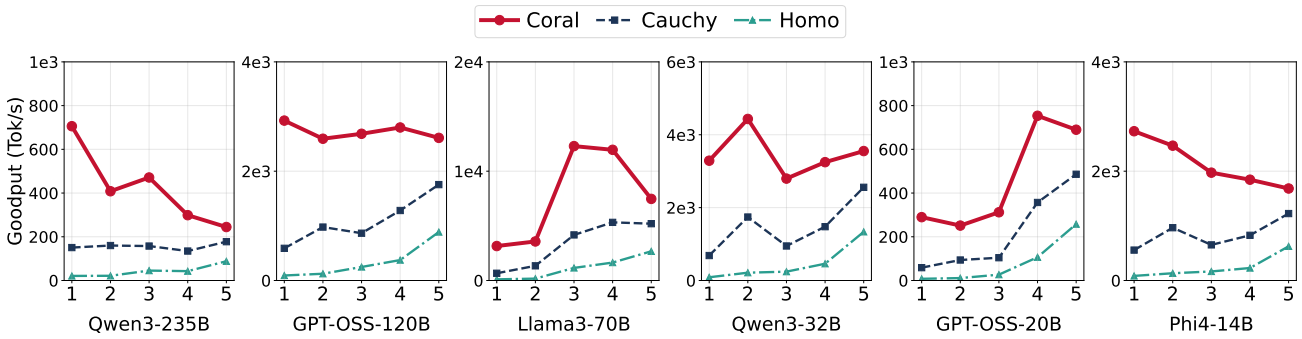


Figure 10. Decode goodput across epochs under tight resource availability (extended setup). Prefill follows the same trend.

the extended setup (Qwen-3 235B, GPT-OSS 120B, Llama-3 70B), observing an average latency deviation of under 2%. Together, these results confirm the simulator's high fidelity as a proxy for our large-scale evaluations.

6.3 End-to-End Cost Efficiency

This section evaluates whether Coral reduces total serving costs compared to state-of-the-art baselines across diverse model mixtures and heterogeneous GPU pools (Fig. 7).

In the core setup (Fig. 7a), Coral reduces average hourly costs by 1.62× over Homo and 1.60× over Cauchy, with the online ILP resolving in 0.24s on average. As the per-model breakdown in Fig. 7b shows, Qwen-3 32B drives roughly 60% of the total expenditure because it is the largest model and lacks the compute-saving MoE or hybrid attention mechanisms. Coral yields its highest savings on this model (2.02× over Homo, 1.88× over Cauchy), primarily during the prefill phase. Examining the selected prefill configurations, both baselines restrict replicas to single nodes (e.g., one replica per 2×L40S or 4×L4 node), varying the node mix across epochs to track demand. Conversely, Coral deploys heterogeneous templates that fuse L4 and L40S nodes into a single replica (e.g., one 1×L4 plus three 1×L40S nodes), employing non-uniform pipeline partitioning and intra-stage data parallelism.

In the extended setup (Fig. 7c), Coral achieves even greater savings, reducing average hourly costs by 3.14× over Homo and 2.66× over Cauchy, with an average ILP solve time of 9.68 seconds. Qwen-3 235B again dominates costs (Fig. 7d) due to its massive parameter count, despite its MoE architecture. Coral reduces Qwen-3 235B serving cost by 4.93× over Homo and 4.08× over Cauchy, again driven by prefill optimizations. A similar architectural pattern emerges: while baselines confine replicas to monolithic 8×A100 or 8×H100 nodes, Coral instead uses templates that combine multiple smaller 1- and 2-GPU A100 and H100 nodes into one replica, optimizing capacity through non-uniform pipelining.

These results confirm that intra-replica heterogeneity is critical for cost efficiency; mixing GPU types allows Coral to tailor instances to specific latency SLOs and throughput demands (Sec. 2.2). Finally, across all methods and scales, amortized initialization overhead remains below 1% of total hourly cost, showing that the allocator's initialization penalty successfully suppresses unnecessary cluster churn.

6.4 Goodput Under Tight Resource Availability

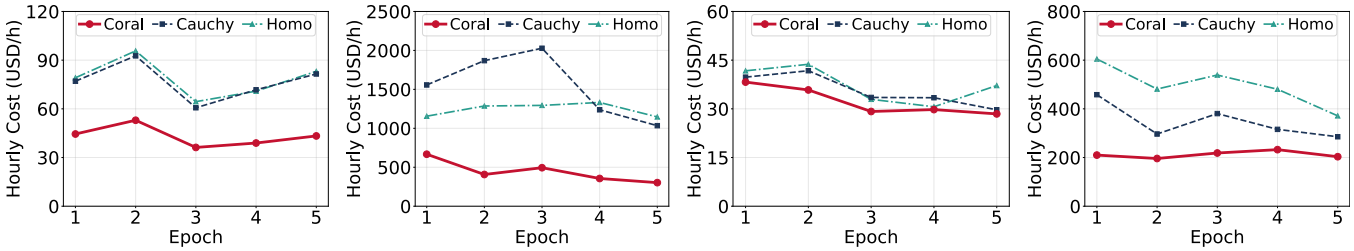
This section evaluates whether Coral sustains high goodput and mitigates contention under tight availability. If a method cannot satisfy all models' throughput demands, we uniformly throttle its per-model arrival rate until a feasible solution exists, preserving the balanced-demand assumption.

In the core setup (Fig. 8a, 9), Coral delivers 1.66× higher average goodput than Homo, reducing the goodput-normalized hourly cost by 1.67×. Against Cauchy, Coral achieves 1.12× higher goodput and a 1.73× reduction in normalized cost. The online ILP resolves in just 0.11 seconds on average.

Compared to its solution in Sec. 6.3, Coral's cluster setup is 10% more expensive on average. During the tightest interval (epoch 1), Coral redirects three high-end L40S GPUs from Phi-4 to Qwen-3—the heavier model that critically requires them—while shifting Phi-4 to A10G and L4 nodes. To achieve this, Coral expands its active heterogeneous templates from 1 to 4. Conversely, the baselines lack cross-model coordination: Homo optimizes each model in isolation, and Cauchy rigidly encodes per-model efficiency in its objective. Consequently, both baselines force all models to contend for the scarce L40S GPUs, leaving none with sufficient capacity.

In the extended setup (Fig. 8b, 10), Coral drastically outperforms the baselines. Against Homo, Coral delivers an overwhelming 7.64× higher average goodput while reducing goodput-normalized hourly cost by 8.99×. Against Cauchy, it achieves 2.39× higher goodput and a 3.27× reduction in normalized cost. The online ILP resolves in 41s on average.

Here, Coral's allocation is 25% more expensive on average than in Sec. 6.3. The tightest interval (epoch 3) also requires the longest solve time (2 minutes) and incurs the largest cost increase (53%). During this epoch, Coral scales its heterogeneous instances from 9 to 19. To sustain throughput for the three largest models (Qwen-3 235B, GPT-OSS 120B, Llama-3 70B), Coral deploys 16 distinct heterogeneous instances—up from 6 in Sec. 6.3, where only Qwen-3 235B relied heavily



(a) Large-Heavy, core setup. **(b)** Large-Heavy, extended setup. **(c)** Small-Heavy, core setup. **(d)** Small-Heavy, extended setup.

Figure 11. Hourly cost under imbalanced demand. 80% of requests are routed to either the largest (Large-Heavy) or smallest (Small-Heavy) third of models. Coral's cost advantage expands significantly when large models dominate the workload.

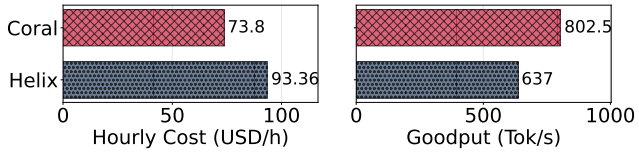


Figure 12. Comparison with Helix on Helix's "High GPU-Heterogeneity Cluster" setup.

on them. This intricate cross-model coordination drives the longer solve time. As in the core setup, the baselines fail because models greedily compete for scarce A100 and H100 nodes, resulting in severe resource starvation.

These results yield two key takeaways: first, global joint optimization is indispensable for satisfying aggregate demand under severe resource scarcity; second, heterogeneous Serving Templates provide the solver with the critical flexibility needed to resolve cross-model hardware contention.

6.5 Robustness to Demand Imbalance

This section evaluates Coral's resilience to skewed throughput demands across models. We consider two imbalanced distributions: *Large-Heavy* routes 80% of requests to the top third of models by size (Qwen-3 32B in core setup; Qwen-3 235B and GPT-OSS 120B in extended setup), while *Small-Heavy* routes 80% to the bottom third (Phi-4 14B in core setup; Phi-4 14B and GPT-OSS 20B in extended setup).

Large-Heavy (Fig. 11a, 11b). In the extended setup, Coral reduces average hourly costs by 2.79× over Homo and 3.47× over Cauchy; in the core setup, it achieves reductions of 1.82× and 1.78×, respectively. In this regime, the large models dominate spending, consuming ~80% of the total budget. Because these massive models require multi-GPU replicas, Coral has ample opportunity to exploit intra-replica heterogeneity, driving large cost savings.

Small-Heavy (Fig. 11c, 11d). In the extended setup, Coral lowers hourly costs by 2.34× over Homo and 1.64× over Cauchy. Despite handling only a fraction of the traffic, Qwen-3 235B and Llama-3 70B are so large that they still account for 40% and 20% of the total cost, respectively, allowing Coral to extract substantial savings from them. However, in the core setup, all three methods yield nearly identical costs, with Coral only ~10% cheaper on average. This is expected: the cost-dominating Phi-4 14B fits on 1–2 GPUs, leaving minimal room for Coral to perform intra-replica optimization.

6.6 Single-Model Comparison with Helix

This section compares Coral against Helix [30], which optimizes model placement for a single model on a **fixed** heterogeneous cluster via MILP.

We adopt Helix's largest experimental setup (the "High GPU-Heterogeneity Cluster") and bias the configuration heavily in Helix's favor. Coral allocates from the exact same cluster Helix uses (4×A100 40G, 6×V100 16G, 16×L4 24G, and 38×T4 16G) to serve the 70B Llama model under AWS US-East-2 pricing. We drive the workload at 4 req/s—exceeding Helix's reported maximum throughput—and constrain Coral with prefill and decode SLOs of 2090 ms and 730 ms, matching the median latencies achieved by Helix. Helix itself runs without SLO constraints. To avoid prohibitive hardware costs, we evaluate Coral using our high-fidelity simulator and compare directly against Helix's reported simulated results.

As Fig. 12 demonstrates, Coral reduces costs by 21% and improves throughput by 26% while strictly satisfying both SLOs. Coral achieves this by dedicating the A100 nodes to prefill and partitioning the remaining hardware into three independent decode instances (combining L4 and T4 nodes), intentionally idling six V100s and one T4 to minimize cost. Conversely, Helix forces all 64 GPUs into a single monolithic pipeline, incurring severe cross-stage communication bottlenecks. By decomposing the resource pool into smaller, right-sized Serving Instances, Coral eliminates the overhead of massive heterogeneous pipelines, delivering higher performance at a lower cost.

6.7 Ablation Study

To isolate the impact of the online resource allocation ILP (Sec. 4.3), we ablate Coral against a *Template Greedy* baseline that independently selects the most cost-efficient Serving Template per model. We evaluate both core and extended

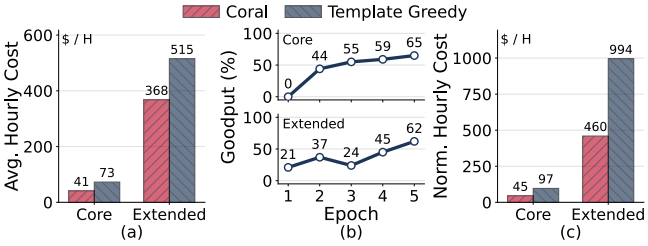


Figure 13. Impact of global allocation. (a) Avg. hourly cost under *abundant availability*. **(b, c)** Goodput and goodput-normalized hourly cost under *restricted availability*.

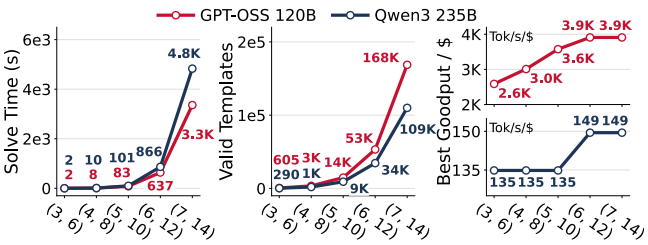


Figure 14. Template pruning sensitivity. Although solve time and template count scale exponentially with pruning parameters $(N_{\max}, \rho)$, best cost efficiency plateaus at $(6, 12)$.

setups under abundant (Sec. 6.3) and restricted (Sec. 6.4) hardware availability. Under abundant availability, Coral's global optimization reduces average hourly costs by 1.78× in the core setup and 1.40× in the extended setup compared to greedy selection (Fig. 13(a)). Under restricted availability, the greedy approach severely degrades, achieving only 45% and 38% of Coral's goodput on average across the two setups, and completely failing to find a feasible solution during one core setup epoch (Fig. 13(b)). Coral reduces the goodput-normalized hourly cost by 2.16× across both setups (Fig. 13(c)). These results confirm that global, cross-model coordination is strictly necessary to navigate hardware contention and maximize cost efficiency.

6.8 Sensitivity of Template Pruning

This section evaluates the sensitivity of the Serving Template generator to its two pruning parameters (Sec. 4.2): the node cap $N_{\max}$ and the memory capacity ratio ρ. Because adding nodes inherently inflates aggregate memory, we sweep both parameters jointly. Using the GPT-OSS 120B prefill and Qwen-3 235B decode phases as a testbed, we execute the generator on a 384-core AWS c8i instance. For each $(N_{\max}, \rho)$ configuration, we measure the total valid templates produced, the aggregate solve time, and the best cost efficiency (Goodput/USD) achieved.

As Fig. 14 demonstrates, while both template count and solve time scale exponentially with $(N_{\max}, \rho)$, maximum cost efficiency plateaus at $(N_{\max}, \rho) = (6, 12)$. This is expected: large node sets suffer from higher inter-node communication overhead and are dominated by partitioning those same resources into multiple smaller, independent replicas. Searching beyond this plateau needlessly inflates offline compute costs without discovering more efficient templates. These results empirically validate the principled pruning strategy proposed in Sec. 4.2 and justify our default configuration of $(N_{\max}, \rho) = (6, 12)$ for all end-to-end evaluations.

7 Related Work

Prefill-Decode (PD) Disaggregated LLM Serving. PD disaggregation provisions prefill and decode independently to exploit their distinct execution characteristics [15, 39, 41, 55, 56]. Splitwise [39] and DistServe [56] pioneer this separation. Cauchy [55] extends it by assigning different GPUs to the prefill and decode pools. However, these systems confine heterogeneity strictly to the phase boundary; individual replicas remain entirely homogeneous. Coral fundamentally extends this paradigm by enabling *intra-replica* heterogeneity via hybrid pipeline and data parallelism, seamlessly supporting both PD-disaggregated and aggregated serving (Sec. 5.1).

Resource Allocation for LLM Serving. Another line of work scales serving resources dynamically against fluctuating demand, pricing, and availability [17, 29, 33]. SpotServe [33] leverages preemptible instances, SkyServe [29] autoscales replicas across multi-cloud regions, and SageServe [17] integrates workload forecasting to minimize reconfiguration overhead. Crucially, while these systems optimize cluster-level provisioning and routing, they treat each replica as a rigid, homogeneous block. Coral operates orthogonally: by jointly optimizing hardware selection and model placement, it unlocks the cost efficiency of intra-replica heterogeneity that prior allocators leave on the table.

Heterogeneous LLM Serving. Prior work leverages GPU heterogeneity across multiple dimensions. Mélange [12] shows that optimal GPU choice depends on workload and SLOs, but restricts replicas to a single homogeneous type. BOute [20] routes requests across fixed heterogeneous hardware to trade off latency and quality. Others optimize composition and placement but minimize offline batch makespan on static pools [19]—the dual of Coral's problem, without online reconfiguration. Systems like Helix [30] and HexGen [21] optimize placement on *fixed* heterogeneous clusters, abstracting away resource provisioning. Similarly, OpWeave [26] jointly optimizes operator partitioning and deployment configurations across heterogeneous GPUs, but assumes a given hardware pool. These approaches do not address dynamic resource provisioning (and as Sec. 4.1 shows, wrapping placement optimization in an online allocation loop is intractable). Coral is the first to jointly provision dynamic heterogeneous resources *and* orchestrate model placement across them for multiple LLMs under strict latency SLOs.

Multi-Model LLM Serving. Recent systems also target the concurrent execution of multiple LLMs. Prism [54] dynamically redistributes memory among colocated models, Aegaeon [50] applies token-granularity autoscaling, and FlexPipe [28] multiplexes models via dynamic pipeline refactoring. While these systems excel at sharing and scheduling within a *fixed* hardware pool, they assume the underlying cluster composition is pre-determined. Coral solves the upstream challenge: determining exactly which heterogeneous resources to provision and how to place models across them. Consequently, these fine-grained runtime mechanisms could be naturally layered atop Coral's Serving Instances.

8 Conclusion

This paper presents Coral, an adaptive, heterogeneity-aware system for cost-efficient multi-LLM serving. Coral jointly optimizes resource allocation and model placement via a lossless two-stage decomposition: offline ILP-based *Serving Template* generation and online template instantiation. Removing the heavy placement search from the critical path accelerates online scheduling from hours to tens of seconds, enabling the cluster to dynamically adapt to fluctuating demands and hardware availability. Evaluated across 6 models and 20 GPU configurations, Coral reduces serving costs by up to 2.79× and delivers up to 2.39× higher goodput under severe resource constraints.

Acknowledgments

Generative AI tools, including Codex, Claude Code, and Gemini, were utilized to assist with code generation during system implementation and to refine the prose of this manuscript. This research was supported in part by NSF awards CNS-2211882 and CNS-2239351, Sloan Foundation faculty fellowships, and research awards from Amazon, Cisco, Google, Jane Street, Meta, NVIDIA, Oracle, Qualcomm, and Samsung. We also thank the members and companies of the PDL consortium (Bloomberg LP, Everpure, Google, Jane Street, LayerZero Labs, Meta, Microsoft Research, Oracle Corporation, Salesforce, Uber, Western Digital) for their interest, insights, feedback, and support. We thank Lambda for providing cloud computing credits that supported the experiments conducted in this work.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

[2] Amazon Web Services. 2023. AWS and Anthropic announce strategic collaboration to advance generative AI. Accessed: 2026-04-20. https://press.aboutamazon.com/2023/9/amazon-and-anthropic-announce-strategic-collaboration-to-advance-generative-ai

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL] https://arxiv.org/abs/2004.05150

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[5] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 8359–8388. https://proceedings.mlr.press/v235/chiang24b.html

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger (Eds.), Vol. 25. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[7] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. arXiv:2406.11931 [cs.SE] https://arxiv.org/abs/2406.11931

[8] Jiaang Duan, Shenglin Xu, Shiyou Qian, Dingyu Yang, Kangjin Wang, Chenzhi Liao, Yinghao Yu, Qin Hua, Hanwen Hu, Qi Wang, Wenchao Wu, Dongqing Bao, Tianyu Lu, Jian Cao, Guangtao Xue, Guodong Yang, Liping Zhang, and Gang Chen. 2025. GFS: A Preemption-aware Scheduling Framework for GPU Clusters with Predictive Spot Instance Management. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Pittsburgh, PA, USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 117–131. doi:10.1145/3760250.3762231

[9] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39. http://jmlr.org/papers/v23/21-0998.html

[10] Google. 2023. Introducing Gemini: Our Largest and Most Capable AI Model. https://blog.google/innovation-and-ai/technology/ai/google-gemini-ai/

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI]

https://arxiv.org/abs/2407.21783

[12] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. Mélange: Cost Efficient Large Language Model Serving by Exploiting GPU Heterogeneity. arXiv:2404.14527 [cs.DC] https://arxiv.org/abs/2404.14527

[13] Gurobi Optimization, LLC. 2026. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[14] Pieter Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., Sebastopol, CA, USA.

[15] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[17] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St. Amant, Chetan Bansal, Victor Ruhle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. 2025. SAGESERVE: Optimizing LLM Serving on Cloud Data Centers with Forecast Aware Auto-Scaling. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 3, Article 61 (Dec. 2025), 24 pages. doi:10.1145/3771576

[18] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[19] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. 2025. Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs. arXiv:2502.00722 [cs.DC] https://arxiv.org/abs/2502.00722

[20] YOUHE JIANG, Fangcheng Fu, and Eiko Yoneki. 2026. BOute: Cost-Efficient LLM Serving with Heterogeneous LLMs and GPUs via Multi-Objective Bayesian Optimization. In *Proceedings of Machine Learning and Systems*, A. Chowdhery and Z. Jia (Eds.), Vol. 8. MLSys, 731–747. https://proceedings.mlsys.org/paper_files/paper/2026/file/ed1d3d4c64dc1b95332a8cde3f2a0bdf-Paper-Conference.pdf

[21] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. 2024. HexGen: Generative Inference of Large Language Model over Heterogeneous Environment. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 21946–21961. https://proceedings.mlr.press/v235/jiang24f.html

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL] https://arxiv.org/abs/2006.16668

[25] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Yi-Hsiang Lai, Xinhao Cheng, Xupeng Miao, and Zhihao Jia. 2026. AdaServe: Accelerating Multi-SLO LLM Serving with SLO-Customized Speculative Decoding. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 36–54. doi:10.1145/3767295.3769315

[26] Zikun Li, Yixuan Mei, Shiqi Pan, Zixuan Chen, Xiaowen Zhang, Mengdi Wu, Shuhuai Lin, Yutong Yang, Zhihao Zhang, Xupeng Miao, Rashmi Vinayak, and Zhihao Jia. 2026. OpWeave: Flexible Operator Disaggregation for Heterogeneous LLM Serving. arXiv:2609.14237 [cs.DC] https://arxiv.org/abs/2609.14237

[27] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic Evaluation of Language Models. arXiv:2211.09110 [cs.CL] https://arxiv.org/abs/2211.09110

[28] Yanying Lin, Shijie Peng, Chengzhi Lu, ChengZhong Xu, and Kejiang Ye. 2026. FlexPipe: Adapting Dynamic LLM Serving Through Inflight Pipeline Refactoring in Fragmented Serverless Clusters. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 72–88. doi:10.1145/3767295.3769316

[29] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. 2025. SkyServe: Serving AI Models across Regions and Clouds with Spot Instances. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 159–175. doi:10.1145/3689031.3717459

[30] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 586–602. doi:10.1145/3669940.3707215

[31] Meta AI. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. Accessed: 2026-04-20. https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[32] Meta Platforms, Inc. 2017. Gloo: A collective communications library. GitHub. https://github.com/pytorch/gloo

[33] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. doi:10.1145/3620665.3640411

[34] Microsoft Corp. 2023. Microsoft and OpenAI extend partnership. Accessed: 2026-04-20. https://blogs.microsoft.com/blog/2023/01/23/microsoftandopenaiextendpartnership/

[35] NVIDIA. 2015. NVIDIA Collective Communication Library (NCCL). GitHub. https://github.com/NVIDIA/nccl

[36] NVIDIA. 2026. *Developing a Linux Kernel Module using GPUDirect RDMA*. NVIDIA Corporation. https://docs.nvidia.com/cuda/gpudirect-rdma/

[37] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[38] OpenAI. 2022. ChatGPT. https://openai.com/index/chatgpt/

[39] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Piscataway, NJ, USA, 118–132. doi:10.1109/ISCA59077.2024.00019

[40] Perplexity Team. 2026. Inside the Rise of Enterprise AI Model-Switching. Accessed: 2026-04-20. https://www.perplexity.ai/hub/blog/inside-the-rise-of-enterprise-ai-model-switching

[41] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[42] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[44] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 1348–1362. doi:10.1109/HPCA61900.2025.00102

[45] Foteini Strati, Zhendong Zhang, George Manos, Ixeia Sánchez Périz, Qinghao Hu, Tiancheng Chen, Berk Buzcu, Song Han, Pamela Delgado, and Ana Klimovic. 2025. Sailor: Automating Distributed Training over Dynamic, Heterogeneous, and Geo-distributed Clusters. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 204–220. doi:10.1145/3731569.3764839

[46] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov,

Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[47] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2* (Toronto ON, Canada) *(KDD '25)*. Association for Computing Machinery, New York, NY, USA, 5831–5841. doi:10.1145/3711896.3737413

[48] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. 2024. Can't Be Late: Optimizing Spot Instance Savings under Deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 185–203. https://www.usenix.org/conference/nsdi24/presentation/wu-zhanghao

[49] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. 2025. The Rise and Potential of Large Language Model Based Agents: A Survey. *Science China Information Sciences* 68, 2 (2025), 121101. doi:10.1007/s11432-024-4222-0

[50] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1030–1045. doi:10.1145/3731569.3764815

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[52] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[53] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[54] Shan Yu, Yifan Qiao, Mingyuan Ma, Yangmin Li, Shuo Yang, Xinyuan Tong, Yang Wang, Zhiqiang Xie, Yuwei An, Shiyi Cao, Ke Bao, Deepak Vij, Xiaoning Ding, Yichen Wang, Qingda Lu, Zhong Wang, Gao Gao, Harry Xu, Junyi Shu, Jiarong Xing, and Ying Sheng. 2026. Prism: Cost-Efficient Multi-LLM Serving via GPU Memory Ballooning. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 75–93. https://www.usenix.org/conference/osdi26/presentation/yu-shan

[55] Yihui Zhang, Han Shen, Renyu Yang, Di Tian, Yuxi Luo, Menghao Zhang, Li Li, Chunming Hu, Tianyu Wo, Chengru Song, and Jin Ouyang. 2026. Cauchy: A Cost-Efficient LLM Serving System through Adaptive Heterogeneous Deployment. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25)*. Association for Computing Machinery, New York, NY, USA, 881–893. doi:10.1145/3772052.3772264

[56] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin